Programming Principles for the Semantics of the Semantics of Programs

Meurig Beynon
Dept of Computer Science
University of Warwick
Coventry CV4 7AL, UK

Abstract

Fundamental concerns of relevance to programming are raised by Brian Smith in [19]. These focus on the relation that holds between a program and the world outside – the semantics of what is referred to by computer scientists as the semantics of the program. This paper explains how programming concepts developed by the author in collaboration with others may contribute to "a theoretical framework that does justice to practice" such as is discussed in [19].

Introduction

The role of logic in AI is the subject of a longstanding controversy dating from McCarthy's Advice Taker (1958). A rich context for the modern debate is provided by the commentaries on a celebrated paper by McDermott [15], in which he explains his reasons for renouncing "logicism". McDermott's paper can be seen as a challenge for programming language theory. It indicates that neither logic nor programming methods that are based upon conventional logical foundations can adequately meet the needs of AI. This sobering conclusion suggests the theme of this paper: what are the limitations of formally-based methods of computer programming, and how can they be remedied in a principled manner outside the conventional formal framework?

The significant implications of the logicism debate for the theoretical foundations of computer programming are brilliantly analysed by Brian Cantwell Smith in [19]. Smith highlights a fundamental distinction between the semantics of a program as it is understood in theoretical computer science and the content relation that AI needs to study, viz the relationship between the computational process and the external world. He explicitly identifies tenets of traditional formal logic that make it unsuitable for describing this relationship. He also makes prescriptive suggestions for an alternative foundation for programming consistent with two "lessons of logic".

This paper reviews, in the light of Smith's analysis, an approach to programming that has been developed under the direction the author over several years. A particular focus is the need identified in [19] for appropriate alternative foundations for programming. The sources for our work [3,4,5,6] include a variety of papers on interactive graphics and visualisation, on the use and development of variables in mathematics and comput-

er science, on modelling and simulating concurrent systems and on software specification. A common theme in these papers is the representation of state information by a family of definitions of variables (a definitive – for definition-based – script, such as is used to define the cells in a spreadsheet). The principal motivating ideas are summarised in [3].

Our prescription for programming, as represented by [3], resembles that expounded by Smith in [18] and [19] in many respects. This is especially interesting because it has been derived independently. Characteristic ideas relevant to [19] include:

- the representations we have used in specifying interaction serve as models of external entities as observed by the user (cf the spreadsheet)
- definitive scripts represent indivisible propagation of state change and can specify content relations such as are discussed in [19]
- definitive scripts are the basis for a new theory of reference in which variables can designate different values in a way that is "formal" in the sense of [19]
- general-purpose programming using definitive representations of computational state is intimately linked with the introduction of agents and agent views.

The first two sections of the paper loosely take the form of a commentary on Smith [19]. A digest of reference [19] is included as an Appendix: this includes three tenets of traditional formal logic (t1, t2, t3) that – according to Smith's thesis – require reconstruction in an adequate theoretical framework for modern programming. All subsequent references to Smith should be understood as citing [19].

The paper is in three sections. §1 discusses the connection between logicism in AI and formal methods of programming based upon traditional logical foundations. §2 will examine the methods we have applied to "1-agent programming" that is an essential ingredient of interaction and design activity. §3 will discuss the application of our methods of representation to multi-agent programming.

1. Applying the Lessons of Logic to programming

1.1. Logicism in the theory of programming

Following Smith, we can view a programming system as having two factors. "The first factor is the representational system's mechanics ... what must be directly realised in a physical substrate if the system is to do any work." In a programming system, the first factor is specified by the computational model. Theoretical computer science has focussed on giving a formal account of the first factor aspect of programs; this is what is meant by programming language semantics.

The second factor is concerned with what the symbols are about, with their interpretation in the logician's sense. "EVALUATE $\Sigma R_i - \Sigma C_j$" describes and prescribes an ab-

stract computation that can be specified and performed without knowing that we are finding the profit from a sale by summing receipts and subtracting costs. The meaning of the program in its relation to the external world is the semantics of the semantics of the program in the terminology of computer science.

The science of programming can be seen as sanctioning as formal a particular kind of computer use that is "programming" in its narrowest sense. To program is to preconceive and prescribe an unambiguous behaviour. This interpretation of programming is defined with reference to the first factor, as in "prescribing the sequence of actions to be performed by an electronic computer" (cf [9]). Specifying a program formally entails preconceiving its execution to such an extent that its behaviour can be described declaratively, that is, using only non-procedural mathematical variables [6].

In formal specification of programs of this nature, the relationship between the program and the external world – its second-factor aspect – is treated as a separate concern. The programmer's task is to organise and represent knowledge of this relationship in the absence of a formal framework. In practice, all programming activity involves the solution of this knowledge representation task. Capturing the semantics of the semantics of programs is what makes programming problems hard.

Smith's insights into formal systems are a useful way to understand historical developments in programming. The idea that a formal account of programming must necessarily be based upon the same foundations as mathematical logic has influenced programming as logicism has influenced AI. "Logicism" in programming is associated with the representation of ever more sophisticated forms of preconceived knowledge about program execution. By way of illustration, ingenious functions can be devised to specify programs with preconceived patterns of interaction (as in the pure functional programming language Haskell [13] §7). Developments of this nature can be viewed as enhancing the computational system the programmer uses. This approach works well when the program requirement is simple and the underlying computational model is orthodox, e.g. when realising a particular input-output relation using a sequential Von Neumann architecture. It is less useful in complex programming applications, e.g. when considering real-time and non-functional requirements for the class of embedded, concurrent and real-time systems described by Pnueli [17] as *reactive*.

Since the early days of programming, describing the first-factor aspect of a program – its operational interpretation – has been simplified out of all recognition by declarative programming techniques directly inspired by the study of programming language semantics. More recently similar techniques have been promoted as a way of simplifying the knowledge representation task for more complex programming applications. According to Smith's thesis, this is a conceptual error: a specification medium formally based on traditional logic can not achieve the essential dependence between first and second factors (cf tenet t2).

1.2. Programming from a post-logicist perspective

Accepting Smith's thesis has important implications for the future of programming. It focusses attention upon the semantics of semantics of programs and upon the need for a new theoretical framework within which to give an account of the second factor aspects of programming. A reappraisal of prejudices that have been established by existing programming theory is in order. As Smith observes, present theories do not do justice to computational practice.

The current theory of semantics of programming languages is biased towards first-factor concerns. This has been a powerful influence over trends in programming language design. The complexity of the semantics of procedural programming systems has been a strong argument against their dominance in the long-term. The semantics of PASCAL is hard to define because things that resemble valid programs have an ambiguous computational interpretation. In contrast, the semantics of Miranda can be described using relatively few mathematical concepts by appealing to a logical theory of higher-order functions for which a generic method of evaluation can be prescribed.

As Smith's analysis makes clear, knowing the semantics of a program and knowing how to deal with the semantics of the semantics of a program are quite different issues. They are commonly confused. Tenet t1 indicates that referential transparency is no virtue where the representation of program content is concerned, but it is frequently cited as a quality that makes programs more intelligible. Procedural programming is deemed unsatisfactory because its semantics are too complex, as if PASCAL programmers were in the habit of writing programs with an obscure operational interpretation.

Smith's first lesson of logic stresses that knowledge of the content relation can not be reduced to issues of form. When contrasting Miranda and PASCAL it is misleading to pretend that intelligibility of programs is the arbiter. For a complex program, (e.g. one that involves a complicated pattern of interaction with the user) inferring what the program is about from its form is typically hard in either case. A related misconception is that a formal notation is to be preferred because automatic reasoning can magically enhance its content.

Computational practice holds many clues to the defects of current theory. In view of tenet t2, it is unsurprising that research problems involving rich interplay between first and second factors are amongst the most challenging open problems:

- how can we write programs that are easy to interpret?
- how do we write interactive programs that adapt to the user?
- how do we integrate requirements analysis and specification?
- how do we classify computer-aided design, where the user introduces knowledge incrementally?

- how do we program a robot to establish a correspondence between the state of an internal program and sensory input?

The ways in which mathematically-based programming languages have been applied in practice are themselves instructive. The interpretative techniques used to execute logic and functional programs invoke concepts that lie outside the scope of classical formal foundations. Two examples are particularly relevant to this paper. The search process of a standard sequential Prolog interpreter interleaves procedural action with dynamic reconfiguration of dependencies between variables. The development of functional programs involves the editing of scripts of definitions of variables representing functions. Both these mechanisms are practically useful ways of linking the execution of a program to the semantics of its semantics.

Some applications have proved difficult to address using declarative methods. These include applications in which human interpretation plays a significant role, such as programming for interactive graphics and user-interfaces, and those that involve communicating state between agents in a variety of ways, such as reactive systems engineering. Because of their declarative nature, conventional formal specification methods define many procedural aspects by default. This is advantageous where the form of the program is immaterial and only its gross behaviour matters. But, as tenet t3 suggests, the challenge of programming for the second factor lies elsewhere – in finding principles to specify a program in a way that reflects its distinctive context.

The history of object-oriented programming is a most interesting case study in the context of Smith's analysis. The original concept, as first conceived in the language Simula in 1967, was that of programming as a restricted form of system description (cf [8] p61). In contrast to other approaches, object-oriented programming in the Simula tradition was motivated by issues intimately connected with the second-factor account of a program. Its technical contribution was the introduction of objects: entities within the program whose external content was explicitly identified in the design process.

The original vision of object-oriented programming – that programming is concerned primarily with the relation between objects in the application domain – in principle gives much more insight into the nature of programming in its full generality than the products of the formal programming tradition. In justifying the method, the difficult problem was to sustain the connection between application and program in a principled manner. This paper argues that basic abstractions other than the object are necessary for this purpose.

As it was, the credibility of object-oriented programming as a second factor modelling tool could only be demonstrated imperfectly. In the 1970s, the work of Parnas and others [16] subverted objects from their second-factor role to become vehicles for modularisation and information hiding in first-factor accounts of programming. The concepts

of classes and inheritance were subsequently added. The end-result was a new idiom of object-oriented programming, as in Smalltalk: a medium with an ambiguous orientation towards both first and second-factors without any clearly defined principle for their association.

The lack of principle in object-oriented programming, as it developed in the 1980s, could not disguise its expressive power. The commercial success of object-oriented design has vindicated the original concept of Simula that developing programs so that their structure reflects second-factor concerns is a powerful principle. But where analogues of naturally occurring mechanisms for interaction in concurrent systems might have been expected to evolve in the philosophical framework of Simula, concurrency in Smalltalk is confounded by the arbitrary nature of the first-factor communication mechanisms. The task of patching this problem for parallel object-oriented languages has fallen to formal semanticists [1] – experts in remedying first-factor defects in programming language design. According to Smith, the first-factor may be recovered in this process, but the marriage of first and second-factor concerns will not survive.

2. Reconstructing the foundations of programming: a proposal

The history of object-oriented programming has an important moral. To deal effectively with the semantics of semantics of programs, it is not enough to build powerful programming systems. Second-factor concepts can not be introduced in an *ad hoc* way. If a programming paradigm is not to degenerate into an obscure mix of first and second-factor modelling techniques, it must be based on principles that are sufficiently clear and prescriptive.

Smith's tenets complement the picture. Principled methods are needed, but these can not be formal in the conventional sense. The development of programs must be more rigorously constrained by the semantics of their semantics: behavioural equivalence is not discriminating enough. This analysis suggests general guidelines to be observed in the programming models we propose:

- a program must be conceived as an ingredient in a larger context. Our assumptions about the context must be represented in the program model.
- there has to be a criterion for determining what is the right program model for a particular context.
- it will not be possible to describe the semantics of semantics of our programs declaratively. An adequate description must involve state.
- intuitions other than those to which conventional foundations appeal must be invoked.

These guidelines are a working hypothesis. The idea that a program may be prescribed by its context should not be seen as violating the first lesson of logic: the irreducibility of content to form. Our program model will be part mechanism, part description of per-

ceived relation to context. Developing a program will correspond to building a model of its context that is faithful to observation subject to abstraction.

2.1. The second factor in user-computer interaction: definitive scripts

Our approach to programming reflects the guidelines set out above. The simple examples below do not illustrate the scope and power of our programming methods – an issue addressed elsewhere [3,4,5]. The aim is to outline the fundamental ideas and explain how it may be possible to develop them in a principled manner. The research we shall review relates programming activity in its broad sense – as it applies for instance to reactive systems development – to "modelling the relation between each programmable component and its environment". This modelling activity is *state-based*: it describes the state of the total system of which the programs form a part. It is also *agent-oriented*: it describes the roles of the agents in the system, i.e. the components that can potentially effect a change of system state. In his discussion of the second factor, Smith asserts that "agents are what matter for semantical connection". This section considers "1-agent modelling" within our paradigm, as defined below.

In the theory of programming, the term "state", like "semantics", is used in its first-factor meaning to refer to the state of an executing program. In contrast, the states we shall consider are external states associated with second-factor concerns. Such a concept of state can not be dissociated from an agent concept: a state is informally defined with reference to simultaneous observations made by an agent. In the sequel, the term "state" will refer to agent-oriented state. State will be represented by variables whose values are to be interpreted as results of observations.

User-computer interaction is a most appropriate setting for a preliminary investigation of state. The principles reviewed below stem from an attempt to reconcile declarative methods of programming with the representation of states such as arise in interactive design. This problem recalls Backus' concern about the lack of history-sensitivity in functional programming [2]; from the perspective of this paper, it has no satisfactory solution.

State-based activities can be classified in a hierarchy according to the number of agents that are privileged to change the system state. In a 0-agent system, the variables define observations that are not subject to change. Conventional formal systems fall into this category. Examples of 0-agent systems include: a pure functional programming system – where evaluating a function is observation without side-effect, and a static picture. The next simplest are 1-agent systems, where one agent is responsible for all the state-changing activity. Examples include: interactive design, conventional single-step debugging, use of a spreadsheet or single-user data-base interaction. More generally, interaction with a deterministic system can be regarded in this way. Multi-agent systems are those in which concurrent change by many agents is possible. Needless to say, the

classification of a system is itself dependent upon the perspective of the human interpreter, but this generalisation is beyond the scope of our present concern.

The spreadsheet illustrates the essential principles we abstract and generalise. Interaction with a spreadsheet establishes states that persist so long as the user is passive: in this sense, the user is the only state-changing agent. The *state* of the spreadsheet here refers to the values that are displayed to the user to interpret after updating; not to the first-factor computational states associated with the mechanisms of the spreadsheet, such as storing and updating values. For instance, we may imagine that the spreadsheet records the explicit values of variables R – representing a resistance and V – representing a voltage, together with a variable I = V/R that denotes a current. We may suppose also that alongside we have a simple circuit comprising a battery and a variable resistor in series.

The spreadsheet provides the user with a state-based model. It should be noted that the variables R, V and I are not logical variables; they can designate more than one value. This concept of *having an identity* yet being capable of attaining different values according to context is a distinctive ingredient of variables outside the scope of traditional formal logic (cf [20] and [6]). The variables R, V and I have a special status because the user agent can interpret them as physical variables observable in the electrical circuit.

Consider simple experiments that can be performed on the electrical circuit. We can measure the resistance r of the resistor, the current i passing through it, the voltage v supplied by the battery. The experimentor will naturally associate different values with the single variable r according to how the resistor is adjusted. In some "mysterious second-factor" sense, it is the *same* resistance that is being measured in different contexts, though its value may change.

Both the spreadsheet and the circuit can be used to generate triples of values (R,V,I) and (r,v,i). There is an intimate relationship between the variables in the spreadsheet and the corresponding attributes of the circuit. This relationship is defined by the correspondence between observations of the spreadsheet and observations of the circuit. To make this precise, we should constrain the user to interact with the spreadsheet in a manner that is consistent with experimentation with the circuit. Changing R is acceptable, for instance, but redefining I as V.R is "meaningless". It is this concept of meaning we propose to use in defining the semantics of semantics of programs.

The nature of "formality" is an issue here. Smith's thesis indicates that the conventional notion of formality can not encompass the semantic relation we hope to define. There are clues to what Smith regards as an appropriate notion of formality (as referenced in cryptic footnotes to [18] and [19]): "formality, in the end, reduces to first-factor notions of physical realisibility". This is entirely consistent with our proposal: that the circuit

be seen as a model certifying a "formal" status for the non-logical variables in the spreadsheet. In understanding this concept of a model, it is important to distinguish between substituting values for a set of variables to satisfy some logical relations (as in traditional model theory) and identifying attributes that have both identity and value and are observed to change values subject to specified functional relationships.

The family of definitions that underlies a spreadsheet is a simple example of a definitive script. More general scripts are specified using a *definitive notation* based upon a richer algebra of values (scalars in the spreadsheet) and operators (arithmetic in the spreadsheet) for the defining formulae. Examples of values for appropriate underlying algebras include lines, points, and sets of points and lines (as in the definitive notation for line-drawing DoNaLD [5]), or windows, locations, display attributes and character strings (as in the definitive notation for screen layout SCOUT [5]).

There is an important distinction between the data types used in underlying algebras and the traditional data types of procedural programming. For meaningful user-computer interaction, the values that variables in the script designate have to relate to an external state-based model that can be observed or conceived by the user. This is quite unlike conventional programming, where the variables typically designate internal values beyond the user's knowledge or concern, that serve to describe the first-factor aspects of the computation.

The choice of underlying algebra can be seen as defining the boundary between first and second factor concerns. In specifying a state-change, the user assumes that – at some level of abstraction – primitive state changing activities will be performed in a way that can not be more explicitly specified. The evaluation of operators in the underlying algebra that is involved in maintaining definitions is computation of this nature. The script itself is a bridge between this (first-factor) internal computation and the user's (second-factor) external interpretation.

The influence of choosing different underlying algebras can be viewed in two ways. It affects what computation can be visible to the user, and what interpretation must be left to the user's imagination. For instance, when we specify a spreadsheet without its tabular interface using a script of scalar definitions, we may take it for granted that the user knows the current values of scalars, without specifying how this knowledge is conveyed through changes in the state of the screen. If instead we adopt a definitive notation such as SCOUT, we can refine the specification of the interface, to express the precise relationship between internal scalar values and the external screen layout.

1-agent modelling with definitive scripts is as an expressive paradigm for specifying user-computer interaction that has been applied to interactive graphics, user-interface specification and visualisation [5]. It has interesting abstract characteristics relating to second-factor concerns mentioned in [19]. These include:

- a more coherent framework for reference and representation
  The relation between language and model is much tighter than in traditional formal logic – cf tenet t3. Definitive variables provide references that are more useful than declarative variables – which only designate a single value, and are more persistent than conventional procedural variables.
- powerful means of modelling non-computable content relations
  The computation that maintains relationships between spreadsheet values is invisible in our programming paradigm. Conceptually, these relationships are "not computed" and propagate instantaneously as indivisible updates of values. Non-computable relationships in the application can be modelled if the current value of a defined variable is always calculated from its defining formulae.
- appropriate methods of representing context-dependence
  Each new definition or redefinition is interpreted with reference to the existing script. Introducing the additional definitions:

  light_white = (I > 0.5); light_red = (0.5 > I > 0.2)

  to the spreadsheet enhances the semantics of the variables (R,I,V) to reflect a new interpretation of the circuit. In this way, "dynamic and contextual factors contribute to the content" – cf tenet t1.

2.2. The second factor in multi-agent systems: agents and privileges

1-agent modelling is a powerful technique for describing the relation between an agent and its environment. As has been illustrated in §2, it represents second-factor concerns by establishing a correspondence between two independent state-based models: one derived from observations of a physical system (e.g. the electrical circuit), the other defined by a definitive script with a protocol for redefinition.

The modelling process can be interpreted as describing certain physical attributes of an object in a way that reflects the result of experiment and observation. In some cases, it can also be seen as expressing how an object can be used as a computational device. As a simple example, consider a script whose definitions represent the relations between the inputs and outputs of gates in a boolean adder. When the user redefines the values at the input gates in the script, the values of the output gates are updated appropriately, so that the script serves as a program to add two numbers.

A definitive model of an adder has more content than a conventional program. By correlating the values of variables in the script with observations of the gates of the adder, the script can be interpreted as representing the physical computation performed by the circuit. The script is not merely *a* program to add two numbers – relative to some convention for observation, it is *the* appropriate model of a particular physical computation.

For the boolean adder, the correspondence between its physical characteristics and its

computational interpretation is relatively easy to establish. When the values of the input gates to the adder are changed, there is a point in time after which it is appropriate to observe the values of the remaining gates – when the electrical effects have propagated fully. Identifying the appropriate observations of the adder can also be viewed as making an assertion about the context in which the adder is used in computation: it is inappropriate to observe the values of gates whilst the electrical state is unstable. All the electrical activity that accompanies a change in the value of an input gate constitutes one indivisible machine operation for the adder as a computational device.

All computation is defined by interpreting observations of physical systems. In conventional user-computer programming, the relation between first-factor activity – the execution of the program – and interpretable observation of its behaviour is described in indirect and implicit ways. In the programmer's view, the form of the indivisible machine operations is rigidly prespecified by the programming language. The programmer conceives indivisible operations at a higher-level of abstraction but can convey this to the user only by restricting the mode of interaction to hide transient inconsistent states. The visible component of the system state is typically described obliquely. For instance, an interactive PASCAL program specifies the current state of the screen as a side-effect of a sequence of **write** statements. The end-product is a program that must be accompanied, whether explicitly or implicitly, by complex conventions about second-factor interpretation.

Such conventions are inadequate for more complex programming applications. In a reactive system, the computational devices are themselves to be designed and programmed; the interaction between agents demands more explicit methods of expressing indivisibility; the nature of the states and observations to be represented is quite different; the modes of communication between agents are more subtle. Principled methods of relating first-factor and second-factor concerns are needed. Our proposal takes the form of a theory of "programs as system descriptions", as in Simula, in which scripts and agents are used to model observations. The definitive model of a boolean adder is a simple example illustrating the principles we hope to generalise.

Smith's first lesson in logic states that content can not be reduced to form; declarative abstractions can not adequately represent a program in relation to its context. Our long-term goal is to show that a principled approach to program content can be based on a suitable theory of observation that presumes procedural intuitions as primitive. In this view, programming is concerned with constructing models of physical systems that describe the context for programmable components and prescribe their behaviour. The primary task of the programmer is to determine what observations of the system need to be analysed to this end; in our modelling framework, representing these observations will entail prescribing the required programs at some level of abstraction. The modelling framework itself is still under development: it will be based upon specifying agent actions using definitive representations of state. In the putative framework, it should be

possible to model observations of an appropriate physical system directly and faithfully in a state-based manner in such a way that the correspondence between the model and observations of the physical system can be verified systematically by experiment.

The primacy of observation is crucial here. Conventional state-based programming systems provide a toolkit from which rich first-factor behaviours can be constructed as coincidental carriers for a second-factor interpretation. The programmer has discretion over the conventions that bind observation and execution. In contrast, in our approach, the programmer is only free to decide what observations of programmable components and their environment are required for specification: the programs are determined in so far as these observations prescribe. Program development and refinement is implicit in the process of modifying and enhancing the set of observations.

Both definitive scripts and agents are abstractions rooted in observation of systems. Scripts specify perceived functional relationships between values that persist in transitions from state to state. In the view of an agent, a state is defined by a set of simultanous observations, and perceived transitions typically involve instantaneous changes to many observations. Identifying what we deem to be appropriate transitions in an agent view is part of the process of analysing the context within which the agent acts. In modelling the opening of a door, we may think of changing the boolean status of variable "open", or of changing the observed position of points on the door relative to the hinge; at another level of abstraction, we may prefer to think not of an instantaneous action upon the entire door, but of interactions between its constituent molecules that propagate across its width. The role of script in this modelling process is to express what, for the purposes of the computational model, most aptly represents the way in which changes to external values impinge upon the agent. In effect, scripts are used to define primitive machine operations in an agent view.

The role of the agent concept in modelling observation is complementary to that of the script. Scripts specify perceived functional relationships between changing values that describe the total effect of an agent action. There are typically other changes in values that are beyond the control of an agent. To illustrate this, consider the electrical circuit introduced in §2. Suppose that a second variable resistor of resistance s is placed in series with resistor r and that the value of s is under the control of an agent (A) independent of our original experimentor (B). In this case, the kind of model that B can build crucially depends upon how the value of s is manipulated by A, and whether B is aware of its value. In the original circuit, as B changes the value of r, so i changes as in one indivisible action subject to the relation i = v/r. In the modified circuit, B may – for instance – be able to observe r and s, recognise that i = v/(r+s), but have no control over the value of s. The observation that changing r has no predictable effect upon s, yet s is subject to change, discloses the presence of another agent.

Faithful modelling of the interaction between state-changing agents in a system is the

fundamental principle behind our approach to programming [3,4]. Faithfulness entails modelling the effect of agent action as we have modelled user-computer interaction: by identifying the functional dependencies that bind together those variables whose values change indivisibly. The nature of the model is also important. A comprehensive abstract description of the behaviour of the system (e.g. in terms of CSP traces [12]) does not serve the intended purpose of binding first and second-factor concerns. The model we require must be constructed in an agent-oriented manner, so that the effect of one agent upon the state of another is specified with reference to agent capabilities and perceptions. Such modelling has both descriptive and prescriptive value – it subsumes programming.

Multi-agent programming using definitive scripts for state representation involves specifying the characteristics of the computational agents in a system. The LSD notation is introduced for this purpose. LSD is used to represent the relations between agent observations and actions. An LSD specification declares what variables are bound to agents, what values agents can perceive, what variables they are conditionally privileged to redefine, and what functional relationships they expect to observe [3,4]. In general, an LSD specification represents the potential for agent action without consideration of synchronisation and communication constraints. Appropriate simulations can be derived subject to adding such constraints to the model [3,4].

The importance of constructing a conceptual model for a reactive system is widely acknowledged. Harel [11] vividly illustrates how such a model can be applied in the analysis of second-factor concerns. Though he stresses that a conceptual model should have a formal mathematical basis from which to derive executable system specifications, his paper is primarily concerned with the semantics of the semantics of such models. It is in respect of the very issues that Harel addresses, such as visualisation, animation, testing, incremental development, guidelines for refinement, that the themes and proposals in this paper are most relevant.

The general principles of concurrent systems modelling and reactive systems prototyping described in this section have been applied in numerous practical examples. Case studies so far prototyped include animations of a vehicle cruise control system and of Harel's digital watch as described by the statechart in [11]. Our experience of agent-oriented modelling with definitive representations of state has been encouraging, but leaves many issues concerning observation and modelling unexplored. A theory of observation such as we have proposed can only be developed and evaluated in the long-term; it can not be justified by citing a few examples. There is another motivation for including such a bold unsubstantiated proposal in this paper: in view of Smith's thesis, it is essential to propose a theory of comparable scope in conjunction with any new programming method – there is otherwise no principled basis for its future development.

Informal confirmation of the quality of our approach comes from many sources. Our

experience shows that programs can be developed in a disciplined way by modelling that is faithful to observation. Principles of program refinement have been demonstrated: behaviour forming part of the context for a computational agent, initially represented abstractly by a script of definitions, is subsequently replaced by systems of actions associated with other agents. Program fragments can be combined simply and are context-sensitive in ways that recall natural language rather than sequential programs.

Connections with other programming paradigms also suggest directions for future development. Scripts are used as a program development tool in functional programming. Powerful programming mechanisms are based upon compiling constraints into functional dependencies, as in a traditional Prolog interpreter [10]. The integration of rule-based and definitive mechanisms in LSD specifications is well-suited to representing "belief revision" and "entailment" in a unified manner.

Conclusions

This paper amplifies Smith's critique of traditional formal logic as a basis for programming, and proposes ideas that accord well with his views concerning an appropriate theoretical framework that does justice to practice. Definitive scripts help to resolve a central difficulty: representing the way in which content propagates "at the speed of logic". They also improve upon object-oriented abstractions in this respect, since content relations transcend object boundaries in extraordinary ways.

The reconstruction of formal logic is a timely task with ramifications beyond the logicism debate. It is a fundamental problem bearing directly on the future development of programming paradigms (cf [2]), data representation techniques (cf [14]) and reactive systems engineering (cf [11]). Resolving the problems identified by Smith is difficult, but will be essential in meeting the challenges posed by new applications and computational mechanisms.

References

1. P America *OOP: a theoretician's introduction* EATCS Bull 29, 1986, 69-84
2. J Backus *Can programming be liberated from the Von Neumann style?* CACM 21(8), pp.613-641, 1978
3. W M Beynon, S. B. Russ, M D Slade, Y P Yung *Programming as modelling: new concepts & techniques* Proc ISLIP'90, Queen's Univ. Kingston, 1990
4. W M Beynon, M T Norris, R A Orr, M D Slade *Definitive specification of concurrent systems* Proc UKIT'90, IEE Conf Publications 316, 1990, 52-57
5. W M Beynon, Y P Yung *Definitive Interfaces as a Visualisation Mechanism* Proc GI'90, Canadian Inf Proc Soc, 1990, 285-292
6. W M Beynon, S B Russ *The development and use of variables in mathematics and computer science* IMA Conf Series **30**, OUP, 1991
7. W M Beynon, S B Russ *The Interpretation of States: a New Foundation for Compu-*

tation? Computer Science RR#207, Warwick University 1992
8. G Birtwistle, O-J Dahl, B Myrhaug, K Nygaard *Simula Begin* 2nd ed., Studentliteratur, Lund, Sweden, 1979
9. Chambers 20th Century Dictionary, 1972 edition
10. W F Clocksin, C S Mellish *Programming in Prolog* Springer-Verlag 1981
11. D Harel *Biting the Silver Bullet: Towards a Brighter Future for System Development* IEEE Computer (to appear Jan 1992)
12. C A R Hoare *Communicating Sequential Processes* Prentice-Hall Int. 1984
13. P Hudak et al *Haskell: A Non-strict, Purely Functional Language* Report Version 1.1, Aug 1991
14. W Kent *Data and Reality* North-Holland, 1978
15. D McDermott *A critique of pure reason* Comput Intell **3** (1987) 151-160
16. D Parnas *On the Criteria to be used in Decomposing Systems* CACM **15** (1972), 1053-1058
17. A Pnueli *Applications of Temporal Logic to the Specification and Verification of Reactive Systems* LNCS 224, Springer-Verlag 1986, 510-584
18. B. C. Smith *The owl and the electric encyclopedia* A I **47** (1991) 251-288
19. B. C. Smith *Two lessons of logic* Comput Intell **3** (1987) 214-218
20. W. A. Woods *Don't blame the tool* ibid, 228-237

Appendix: A digest of Cantwell Smith's Lessons of Logic

This section reviews the fundamental concepts and ideas introduced in Smith's paper on "lessons of logic" [20]. It is largely composed of edited extracts from [20].

We can view a symbol system as having two factors. The first factor is the representational system's mechanics: what must be directly realised in a physical substrate if the system is to do any work. The second factor is concerned with what the symbols are about, with their interpretation in the logician's sense. In mathematical logic, proof theory gives an account of the first factor, and model theory an account of the second.

The first factor account of a symbol system concerns the form of the symbols, the ways in which they can be composed and decomposed and the operations defined upon them. Every symbol system has a first factor aspect. What really matters about a symbol system is the second factor aspect: the content of the symbols. The first lesson of logic is that content can not be reduced to form. There is more to a symbol system than can be gleaned from its rules and representations.

Second-factor properties of a symbol system are more mysterious than first-factor properties. The content of a symbol isn't in general an intrinsic property of it, but arises as a relation between the system and some other domain. To appreciate the second factor you have to go outside the system, to see how it is connected to, and used in, its environment. Content relations aren't computed: the content relation just *is*, and doesn't

need physical realisibility. It seems that agents are what matter for semantical connection.

We don't know how reference and content work, but we know that they do work, and that there is more to it than proof theory. The job of semantics is to explain, as systematically and rigorously as possible, the interplay between first factor properties and the more elusive second-factor properties of meaning and content.

The second lesson of logic is that first and second factors must be related, despite being conceptually distinct. In mathematical logic, this relationship is established by completeness proofs and by such notions as soundness and validity. A single unified theory must take account of both factors.

Smith complements these lessons by claiming that in human thought processes "first and second factors are constantly and intimately related". He then cites three tenets underlying classical logic that are incompatible with computational practice [that is, with] what are programs actually do, not what we *say* about them:

t1. use can be ignored. A sentence must represent its whole content explicitly.
t2. locally first and second factors can be treated independently, even though they must ultimately be globally related. From step to step, in a formal proof, the first-factor inference procedure can not depend on or affect second-factor semantic interpretation.
t3. language and modelling are categorically distinct types of representation. The linguistic reference relation is non-transitive, but modelling is transitive and "free" in the sense that you are allowed to use a model of X in place of X itself.

In contrast to 1: in natural language, context may implicitly contribute to the content. In contrast to 2: even war-horse programming languages are best understood in terms of locally intertwined factors. With reference to 3: promiscuous modelling is unhelpful in answering fine-grained questions about control, intensional identity, and the use of finite resources. Moreover, current computational systems involve representational structures of all kinds, ranging continuously from linguistic expressions to virtually iconic isomorphisms like bit maps and simulation structures.

Being functionally dependent and being synchronised in time are independent concepts: consider paper, stone, scissors. Where does causal connection fit in?